SVChecker: A deep learning-based system for smart contract vulnerability detection

Ye Yuan[#a], Tongyi Xie*[a,b]
[a]School of Computer Science and Technology, Beijing Institute of Technology, Beijing 10081, China; [b]Academic Affairs Office, Guangxi College of Education, Nanning, Guangxi 530023, China

ABSTRACT

The detection of smart contracts vulnerability is a valuable research problem because smart contracts hold a huge amount of cryptocurrency. In the past, popular detection tools were mainly based on some traditional techniques such as fuzzing and symbolic execution, which rely on fixed expert features or patterns and often miss many vulnerabilities. Recent machine learning approaches alleviate this issue but do not notice the semantic information in the source code. In this paper, we develop a system called SVChecker to classify the smart contract source code written in Solidity. To show the superiority of our system, we conduct experiments on more than 40,000 smart contracts collected from Ethereum. Empirically, our experimental results demonstrate that our system outperforms all popular detection tools.

Keywords: Solidity, vulnerability detection, deep learning

1. INTRODUCTION

Since Satoshi Nakamoto first proposed the concept of Bitcoinin 2008[1], decentralized cryptocurrencies have begun to flourish and have attracted more and more people's attention. Cryptocurrency is a digital currency, what's more, which is not controlled by the central bank but decentralized through blockchain technology. That means users can maintain shared data through a specific consensus protocol in the cryptocurrency network, thereby achieving secure transactions. In recent years, the use of blockchain technology has surpassed peer-to-peer transactions, which is inseparable from the application of smart contracts.

Smart contracts are programs running on blockchains and can perform trusted transactions without a third party[2]. Ethereum is the most popular public blockchain platform for running smart contracts[3]. Besides, due to the immutable nature of blockchain, once a smart contract is deployed on the blockchain, it cannot be modified. Nowadays, more and more transactions on Ethereum are executed automatically through smart contracts. Not only that, with the help of smart contracts, we can develop various types of decentralized apps on Ethereum.

However, because of the enormous economic value brought by smart contracts, it has also attracted the attention of attackers. The solidity language is the most common high-level language used to write smart contracts on Ethereum[4]. Programmers write smart contracts, and programmers cannot guarantee that their code will be executed without vulnerabilities. Coupled with the unmodifiable nature of smart contracts, this leads to a tremendous economic threat once smart contracts are attacked. Therefore, more and more researchers are diving into detecting vulnerabilities in smart contracts. In this paper, we propose a deep learning-based system for vulnerabilities detection of smart contracts written in Solidity language.

Contributions. Our contributions are:

- We propose a method to extract specific code snippets from Solidity source code and label them malicious or benign. This kind of code snippet can focus on the data flow of a particular variable. Also, our approach can help generate a dataset that is more suitable for deep learning model training. We release that dataset at the link below.

- We present the design and implementation of a deep learning-based smart contract vulnerabilities detection system, called Solidity Vulnerability Checker, for source code written in Solidity language. This system can extract the code

#yuanyemse@gmail.com; *28922111@qq.com

International Conference on Computer Application and Information Security (ICCAIS 2021), edited by Yingfa Lu, Changbo Cheng, Proc. of SPIE Vol. 12260, 122600W · © The Authors. Published under a Creative Commons Attribution CC-BY 3.0 License · doi: 10.1117/12.2637775

snippets we defined and check the code snippets have vulnerabilities or not. To facilitate future research, our implementations are released at https://github.com/yesmola/SVChecker.

- We evaluate the results of our system with the public smart contracts dataset Smartbugs and Smartbugs-wild[5] in two ways. On the one hand, we compared our approach with directly using the entire source code as input. On the other hand, we demonstrate that SVChecker outperforms the existing tools, achieving a higher precision, compared to tools including Oyente[6], Securify[7], Slither[8], and Smartcheck[9].

2. RELATED WORK

Existing work on smart contract vulnerability detection can be divided into two categories: conventional detection methods and machine learning-based methods. For conventional detection methods, Contract Fuzzer[10] identifies vulnerabilities by fuzzing and runtime behavior monitoring during execution. Oyente is one of the representatives of symbolic execution tools. The common feature of such tools is that they have poor detection effects for new types of vulnerabilities. For machine learning-based methods, Zhuang et al.[11] introduced a novel temporal message propagation network (TMP) and a degree-free GCN (DR-GCN) to automatically detect smart contract vulnerabilities. Eth2vec[12] is a machine learning-based static analysis tool for detecting code rewriting attacks particularly. And it uses Ethereum Virtual Machine bytecodes as input rather than Solidity source code.

3. DESIGN OF SVCHECKER

In this section, we present the Solidity Vulnerability Checker (SVChecker). Our objective is to design a vulnerability detection system for smart contracts written in Solidity Language. Our system takes a smart contract source code as input and then tells whether it is vulnerable or not. The overview of the proposed system is illustrated in Figure 1, which consists of two phases: (a) training phase; and (b) detection phase. In detail, the SVChecker can be divided into three core modules: (1) code snippets extraction; (2) deep learning model; and (3) detector for unknown source code. In the following content, we will give a detailed explanation of the functions of these three core modules.

3.1 Code snippets extraction

We represent the source code as vectors that can include more contextual semantic relations. However, directly using the entire source code is not a good choice because there is much irrelevant information. To make our system do well, we first propose transforming programs into a representation of *code snippets*. We observe that two reasons cause most vulnerabilities in smart contracts: (1) incorrect operations of variables, like *integer overflow*; (2) improper use of API function calls, like *timestamp dependency* and *reentrancy*. For example, incorrect add operations to a *uint* type variable may cause *integer overflow* and improper uses of *call.value* (a Solidity API) may cause *reentrancy*. In addition, we believe that the statements of *Library* and *Event* in Solidity source codes will not cause vulnerabilities, so we just ignore them.

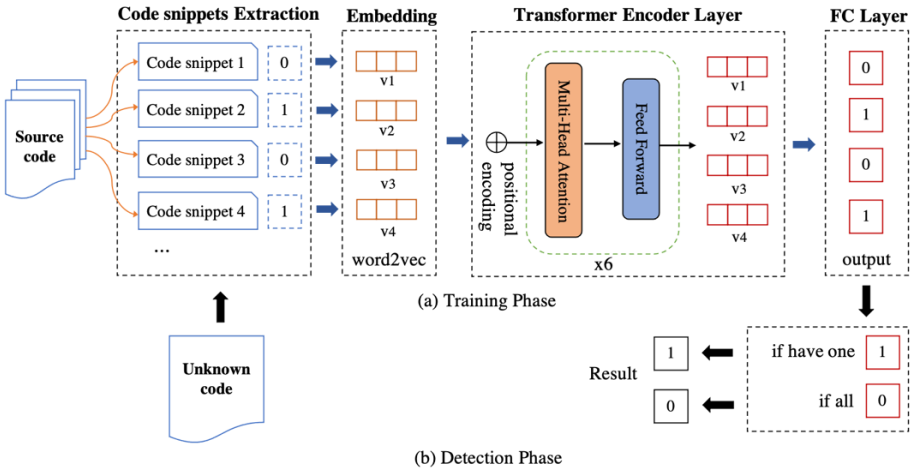

Figure 1. Overview of SVChecker.

Based on this fact, we design the work flow of code snippets extraction. Figure 2 shows an example of that. Firstly, we extract multiple program slices for each global variable and its corresponding functions. In a sense, this kind of program slice can reveal more semantic information. Secondly, we generate the code snippets from the extracted program slices. We notice that most functions and variables' identifier naming is meaningless and has nothing to do with vulnerability. Therefore, we normalize the program slices by mapping user-defined variables and functions to symbolic names (e.g., "VAR1", "FUNC1") one by one. Thirdly, we label the vulnerable code snippet as "1" and label the non-vulnerable code snippets as "0". If a code snippet contains a vulnerable line, it is defined as vulnerable; otherwise, it is non-vulnerable. The data of vulnerable lines can be found in the training dataset.

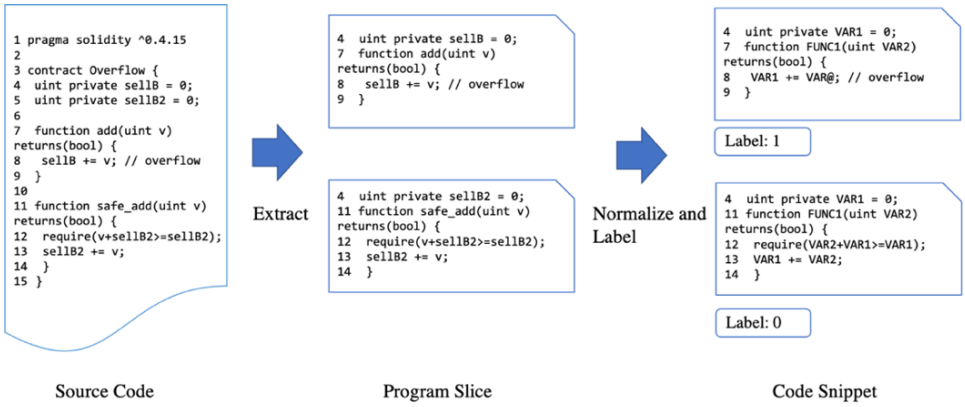

Figure 2. An example of work flow of code snippets extraction.

3.2 Deep learning model

We adopt Word2Vec[13] for the extracted code snippets to encode their tokens into feature vectors. The reason why we choose Wrod2Vec is its widespread use and excellent performance in the field of text classification[14]. It can convert a token to a fixed-length vector. Moreover, we set a hyperparameter τ as the fixed size of vectors corresponding to code snippets for subsequent model training. When the number of tokens in the code snippet is greater than τ, it will be truncated; otherwise, it will be filled with 0.

To learn richer contextual information in the code snippets, we introduce a model based on Transformer-Encoder as the next stage. The encoder is composed of six identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a position-wise fully connected feed-forward network[15]. Transformer has been proven to achieve surprising results on multiple natural language processing tasks[16]. Source code vulnerability detection can also be seen as an NLP task. After that, we use a fully connected layer to map high-dimensional vectors to low-dimensional ones for the final classification task.

3.3 Detector for unknown source code

Our system aims to determine whether the unknown Solidity source code is vulnerable. So, after the training phase, we design the detection phase. During the detection phase, we use an unknown Solidity source code as input. Same as the training phase, this source code is also extracted into multiple code snippets. Each of them will be tested through the trained neural network and then get a result. In other words, one program will get a list of results. If all the results are non-vulnerable, the source code is non-vulnerable. But even if there is only one vulnerable result, the source code is vulnerable.

4. EXPERIMENTS AND RESULTS

In this section, we empirically evaluate our system on two public datasets, namely Smartbugs and Smartbugs-wild. Our experiments focus on answering the following three research questions (RQs):

- RQ1: Can the Code Snippets Extraction module make SVChecker do a better job?
- RQ2: How effective is SVChecker when compared with other vulnerability detection tools forsmart contracts?

- RQ3: Can SVChecker deal with multiple types of vulnerabilities at the same time?

4.1 Experimental settings

Dataset description. The first step of our experiment is to collect enough data. We use two datasets, including Smartbugs and Smartbugs-wild.1) Smartbugs. Smartbugs is a curated dataset that contains 143 annotated contracts with 208 tagged vulnerabilities that can be used to evaluate the accuracy of analysis tools. 2) Smartbugs-wild. Smartbugs-wild is a dataset with 47,398 unique contracts from the Ethereum network. But these smart contacts are not labelled. To solve this problem, we use Oyente to pseudo-label this dataset.

Implementation details. All experiments are conducted on a computer equipped with an Nvidia GeForce RTX 3090 GPU. The code snippets extraction is implemented with C++ (GCC version 9.2), while the neural networks are implemented with Python 3.7 and Pytorch 1.7.0. For training, we use 40,000 smart contracts from Smartbugs-wild and split them into 80% as a training set, 20% as a validation set. For evaluation, we use 4,000 smart contracts from Smartbugs-wild and all smart contracts from Smartbugs as the testing set. We use the Adam optimizer and cross-entropy for the classification loss. More details of experimental parameters can be found in our released source code.

4.2 Performance comparison

4.2.1 Experiments for answering RQ1. In order to evaluate how effective the code snippets extraction module is. We conduct two experiments on the Smartbugs-wild dataset respectively. One contains the code snippets extraction module. Besides, for the sake of controlling the number of code snippets and maintaining the balance between positive and negative samples, we limit the generation of negative samples (i.e., do not contain vulnerabilities). In total, we extracted 62,523 code snippets from 40,000 source codes, consisting of 32,362 positive samples and 30,161 negative samples. The other just use the source code files directly.

We use the widely used Precision, Recall, F1-score, and Accuracy metric as the evaluation. As shown in Table 1, the code snippets extraction module can make the SVChecker more effective. The improvement in each of the metrics is substantial: 1.2% in Precision, 7% in Recall, 4% in F1-score, and 1.8% in Accuracy. Confusingly, our two experiments detected different numbers of positive and negative samples for the same 4,000 samples. In order to find out this reason, we conducted an in-depth analysis.

Table 1. Performance comparison of SVChecker with code snippets extraction or not.

System	Number (P | N)	Precision	Recall	F1	Accuracy
SVChecker (without CSE)	4,000 (2,841 | 1,159)	91.36%	90.29%	90.90%	92.53%
SVChecker (with CSE)	4,000 (2,235 | 1,765)	**92.53%**	**97.23%**	**94.82%**	**94.35%**

Finally, we observe that the Oyente tool incorrectly judged the *add* function in *SafeMath* Library as vulnerable (Figure 3). However, our code snippets extraction module will not extract these pieces of code at all. This fact shows that this module in our system is very useful from another point of view.

```
library SafeMath
{
   function add(uint256 a, uint256 b) internal pure returns (uint256 c)
   {
        c = a + b; // possible bug
        assert(c >= a);
        return c;

    }

}
```

Figure 3. Possible bug found by Oyente.

4.2.2 Experiments for answering RQ2 and RQ3. In order to answer RQ2, we compare the effectiveness of SVChecker with other vulnerability detection tools. At the same time, we use the Smartbugs dataset to test these tools. This dataset provides a collection of vulnerable Solidity smart contracts according to the DASP taxonomy and contains ten different types of vulnerabilities. We select six types of vulnerabilities for testing shown in Table 2. The first three types of vulnerabilities appeared in our training set, while the latter three did not. In this way, we can also answer RQ3. We use

the report provided by Durieux[5] as the results of Oyente, Securify, Slither and Smartcheck. We illustrate the performance of different tools in Table 2. It should be pointed out that the number of samples is different because of the upgrade of Smartbugs.

Table 2. Performance of different tools on the Smartbugs dataset.

Category/tools	Oyente	Securify	Slither	Smartcheck	SVChecker
Reentrancy	5/8 | 62%	5/8 | 62%	7/8 | 88%	5/8 | 62%	**31/31 | 100%**
Arithmetic	12/22 | 55%	0/22 | 0%	0/22 | 0%	1/22 | 5%	**13/15 | 87%**
Time manipulation	0/5 | 0%	0/5 | 0%	2/5 | 40%	1/5 | 20%	**4/5 | 80%**
Access control	0/19 | 0%	0/19 | 0%	4/19 | 21%	2/19 | 11%	**10/18 | 56%**
Unchecked Low-level call	0/12 | 0%	3/12 | 25%	4/12 | 33%	4/12 | 33%	**52/52 | 100%**
Front running	0/7 | 0%	2/7 | 29%	0/7 | 0%	0/7 | 0%	**2/4 | 50%**

As we can see, the SVChecker has the best experimental result in detecting various types of vulnerabilities. And detection rate is far ahead of other detection tools. We make following observations. First, the deep learning-based system SVChecker outperforms the other pattern-based static analysis detection systems. Second, for the types of vulnerabilities that Oyente can detect, the detection rate of the SVChecker exceeds 80%. Even for the types of vulnerabilities that have not appeared in the training set, the SVChecker has a certain detection rate. We think it benefits from the powerful learning ability of the Transform-Encoder. In contrast, the detection effects of other detection tools perform poorly on Smartbugs.

5. CONCLUSION

In this paper, we present the SVChecker system for smart contract vulnerability detection. We propose a practical method to extract specific code snippets from the source code, and this method can help the neural networks perform better. We demonstrate that the SVChecker achieves high accuracy in detecting vulnerabilities compared to existing popular detection tools and can deal with different vulnerabilities.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China (No. 61962005) and Guangxi University Young and middle-aged teachers' basic scientific research ability improvement project (No. 2021KY1934).

REFERENCES

[1] Nakamoto, S., "Bitcoin: A peer-to-peer electronic cash system [EB/OL]," (2009). https://bitcoin.org/bitcoin.pdf
[2] Zou, W., Lo, D., Kochhar, P. S., Le, X. B. D. and Xu, B., "Smart contract development: Challenges and opportunities," IEEE Transactions on Software Engineering, 47(10), 2084-2106 (2019).
[3] Wood, G., "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 151, 1-32 (2014).
[4] Dannen, C., [Introducing Ethereum and Solidity], CA: Apress, Berkeley, (2017).
[5] Durieux, T., Ferreira, J. F., Abreu, R. and Cruz, P., "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," 530-541 (2019). arXiv:1910.10601
[6] Luu, L., Chu, D. H., Olickel, H., et al., "Making smart contracts smarter," 2016 ACM SIGSAC Conf., 254-269 (2016).

[7] Tsankov, P., Dan, A., Drachsler-Cohen, D., et al., “Securify: Practical security analysis of smart contracts,” Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security, 67-82 (2018).
[8] Feist, J., Greico, G. and Groce, A., “Slither: A static analysis framework for smart contracts,” 2019 IEEE/ACM 2nd Inter. Work. on Emerging Trends in Software Engineering for Blockchain (WETSEB), 8-15 (2019).
[9] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., et al., “SmartCheck: Static analysis of ethereum smart contracts,” IEEE 1st Inter. Work. on Computer Society, 9-16 (2018).
[10] Jiang, B., Liu, Y. and Chan, W. K., “ContractFuzzer: Fuzzing smart contracts for vulnerability detection,” 33rd ACM/IEEE Inter. Conf. on Automated Software Engineering (ASE), 259-269 (2018).
[11] Zhuang, Y., Liu, Z., Qian, P., et al., “Smart contract vulnerability detection using graph neural network,” Twenty-Ninth Inter. Joint Conf. on Artificial Intelligence and Seventeenth Pacific Rim International Conference on Artificial Intelligence, 3255-3262 (2020).
[12] Ashizawa, N., Yanai, N., Cruz, J. P., et al., “Eth2Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts,” 47-59 (2021). arXiv:2101.02377v2
[13] Church, K. W., “Word2Vec,” Natural Language Engineering, 23(1), 155-162 (2017).
[14] Lilleberg, J., Yun, Z. and Zhang, Y., “Support vector machines and Word2vec for text classification with semantic features,” IEEE Inter. Conf. on Cognitive Informatics & Cognitive Computing,136-140 (2015).
[15] Vaswani, A., Shazeer, N., Parmar, N., et al., “Attention is all you need,” 5998-6008 (2017).
[16] Wolf, T., Debut, L., Sanh, V., et al., “Transformers: State-of-the-art natural language processing,” Proc. of the 2020 Conf. on Empirical Methods in Natural Language Processing: System Demonstrations, 38-45 (2020).